Assignment 4: Heap Data Structures: Implementation, Analysis, and Applications

Name: Pranavi Balakulla
Course: Algorithms and Data Structures
Date: 02/15/2026

Heapsort Implementation and Analysis

1. Heapsort Implementation

Heapsort is a comparison-based sorting algorithm that arranges data using a binary heap structure. It uses a max-heap to efficiently identify the largest element and place it in its correct position within the sorted sequence. In this approach, a Python list is used to represent the heap, taking advantage of the mathematical relationships between parent and child indices.

This algorithm has two main phases:

- **Heap Construction:** Transforming the input array into a max-heap
- **Sorting Phase:** Extracting the maximum element and restoring property of heap repeatedly.

2. Analysis of Heapsort Implementation

2.1 Time Complexity Analysis

Heapsort is a comparison-based sorting algorithm that employs a max-heap data structure to organize elements efficiently. The algorithm proceeds in two main stages: first, constructing the heap, and second, repeatedly extracting the maximum element to build the sorted sequence.

Heap Construction Phase

In the first phase, the input array is converted into a max-heap using a bottom-up approach. Heapification starts at the last non-leaf node and moves upward toward the root. While the heapify operation has a worst-case time complexity of $O(\log n)$ for a single node, the overall cost of constructing the heap is $O(n)$.

This efficiency arises because most nodes are near the leaves and require only a small, constant amount of work. Specifically, the number of nodes at height h in a binary heap is at most $\lceil n / 2^{(h+1)} \rceil$, and the cost of heapifying each node is proportional to h. Summing these costs across all heights results in a total complexity of $O(n)$, making the bottom-up heap construction phase linear in time.

Sorting Phase

Once the max-heap has been constructed, the sorting phase proceeds by repeatedly removing the root element, which holds the maximum value, and placing it at the end of the array. Each extraction involves the following steps:

- Swapping the root element with the last element of the heap, an operation that requires O(1) time.
- Restoring the heap property by applying the heapify operation, which has a time complexity of O(log n) due to the height of the heap.

Because the extraction process is repeated n−1 times, the overall time complexity of the sorting phase is O(n log n).

Overall Time Complexity:

Considering both phases together, Heapsort attains a consistent time complexity of O(n log n) in all cases.

Case	Time Complexity
Best Case	O(n log n)
Average Case	O(n log n)
Worst Case	O(n log n)

This consistency results from the balanced structure of the heap, which is maintained regardless of the initial order of the input. Each extraction involves restoring the heap property along a path whose length is logarithmic in the number of elements, ensuring that Heapsort consistently performs O(n log n) operations.

1. Heap Construction — O(n)

The input array is first converted into a max-heap using a bottom-up heapify process. Although heapifying a single node can take O(log n) time, most nodes are located near the bottom of the tree and require only a small number of operations.

Formally, nodes at height h are at most ⌈n / 2^(h+1)⌉ and cost O(h) each. The total work becomes:

Therefore, building the heap takes O(n) time regardless of input order.

$$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot O(h) = O(n)$$

2. Repeated Extraction — O(n log n)

After the heap is built, the algorithm repeatedly removes the maximum element:

1. Swap the root with the last element — O(1)

2. Restore heap property using heapify — O(log n)

This extraction happens n − 1 times, and each heapify operation travels at most the height of the heap:

(n - 1) * O(log n) = O(n log n)

3. Total Running Time

O(n) + O(n log n) = O(n log n)

4. Why It Is the Same for Best, Average, and Worst Cases

Unlike algorithms such as Quicksort, Heapsort's behavior does not depend on the initial order of the data:

1. The heap is always a complete binary tree → height always log n
2. Every element must be extracted exactly once
3. Each extraction always requires heapify of up to log n levels

Since the number of operations performed is fixed and independent of input arrangement, the running time remains:

O(n log n) for best, average, and worst cases

This consistency is the defining advantage of Heapsort: predictable performance regardless of whether the array is sorted, reverse-sorted, or random.

2.2 Space Complexity Analysis

Heapsort is an in-place sorting algorithm, requiring no additional data structures proportional to the input size, which results in an auxiliary space complexity of O(1).

If the heapify operation is implemented recursively, the recursion depth may reach the height of the heap, leading to a call stack space of O(log n) in the worst case. However, using an iterative implementation of heapify removes recursion entirely, maintaining the true O(1) auxiliary space complexity.

2.3 Additional Overheads

Although Heapsort guarantees O(n log n) time complexity and operates in-place, it introduces certain practical overheads. The heapify procedure requires multiple comparisons and swap operations at each level of the heap. While these operations do not change the asymptotic time complexity, they increase the constant factors involved, making Heapsort less efficient in practice compared to algorithms such as Quick Sort.

Additionally, Heapsort demonstrates non-sequential memory access patterns during heapify operations. Since elements are frequently accessed across different levels of the heap, cache performance may be less efficient compared to algorithms such as Merge Sort or Quick Sort, which typically access memory in a more sequential manner. Consequently, despite its strong theoretical guarantees, Heapsort may not always achieve the fastest performance in practical implementations.

2.4 Summary

The Heapsort algorithm guarantees O(n log n) time complexity in the best, average, and worst cases, making it a predictable and reliable sorting method. It sorts elements in-place with minimal auxiliary memory requirements and ensures correctness through systematic heap construction, repeated extraction of the maximum element, and restoration of the heap property. These features make Heapsort especially suitable for applications that demand consistent performance and low memory usage.

3. Comparison of Heapsort, Quick Sort, and Merge Sort

3.1 Experimental Setup

To evaluate the performance of Heapsort in comparison with Quick Sort and Merge Sort, each algorithm was implemented in Python and tested under the same system environment. Running time was measured using the time.perf_counter() function, which offers precise, high-resolution timing.

For each experimental setup, the algorithms were executed multiple times, and the median execution time (measured in seconds) was calculated. Using the median value helped reduce the impact of system variability and background processes on the results.

The algorithms were tested with various data set sizes and distributions as follows:

Input Sizes:

- 1,000
- 2,000
- 5,000
- 10,000
- 20,000

Input Distributions:

- Sorted (Ascending Order)
- Reverse – Sorted (Descending Order)
- Random

The sorted output is verified using Python's built-in sorted() function.

```
def run_benchmarks(sizes, dists, repeats=5):
    algorithms = [
        ("Heapsort", heapsort),
        ("Quicksort", quicksort),
        ("MergeSort", mergesort),
    ]

    results = []  # list of dict rows
    for n in sizes:
        for dist in dists:
            data = generate_data(n, dist, seed=123)  # same base data per (n, dist)
            for name, fn in algorithms:
                t = time_one(fn, data, repeats=repeats)
                results.append({
                    "n": n,
                    "distribution": dist,
                    "algorithm": name,
                    "median_seconds": t
                })
    return results


def print_table(results):
    # simple grouped print
    results_sorted = sorted(results, key=lambda x: (x["n"], x["distribution"], x["algorithm"]))
    print(f"{'Input Size':<12}  {'Distribution':<15}  {'Algorithm':<15}  {'Execution Time(s)':<20}")
    print("-" * 65)
    for r in results_sorted:
        print(f"{r['n']:<12}  {r['distribution']:<15}  {r['algorithm']:<15}  {r['median_seconds']:<20.6f
```

```python
def heapsort(arr):
    a = arr[:]  # work on a copy for fair comparisons
    n = len(a)

    def heapify(n, i):
        # iterative heapify (sift down) to keep aux space O(1)
        while True:
            largest = i
            l = 2 * i + 1
            r = 2 * i + 2

            if l < n and a[l] > a[largest]:
                largest = l
            if r < n and a[r] > a[largest]:
                largest = r

            if largest == i:
                break
            a[i], a[largest] = a[largest], a[i]
            i = largest

    # Build max-heap: O(n)
    for i in range(n // 2 - 1, -1, -1):
        heapify(n, i)

    # Extract elements: n times * O(log n)
    for end in range(n - 1, 0, -1):
        a[0], a[end] = a[end], a[0]
        heapify(end, 0)

    return a


def mergesort(arr):
    # returns a new sorted list
    n = len(arr)
```

3.2 Experimental Results

The benchmark results revealed the following runtime trends, based on median execution times measured in seconds.

Across all input sizes and data distributions, several consistent patterns were observed.

Performance on Random Inputs:

For random input distributions

- At n=1000, Quick Sort (0.001893s) was the fastest algorithm, followed by Merge Sort (0.002275 s), and Heapsort (0.004721 s).
- At n=20,000, Quick Sort (0.041933 s) has the lowest execution time, followed by Merge Sort (0.067531 s) and Heapsort (0.076126 s).

Against all the input sizes, Quick Sort consistently performed best on random inputs, while Heapsort consistently exhibited higher execution times compared to both alternatives.

Performance on Sorted Inputs:

For already sorted inputs, similar performance patterns were observed.

- At n=5,000, Merge Sort (0.009320 s) achieved the fastest execution time, followed by Quick Sort (0.012300 s) and Heapsort (0.021008 s).
- At n=20,000, Merge Sort (0.038588 s) continued to outperform Quick Sort (0.048058 s), while Heapsort (0.080534 s) once again recorded the highest execution time.

These findings indicate that Merge Sort maintains strong and consistent performance on ordered data. Quick Sort's running time remained close to O(n log n), suggesting that worst-case partitioning was avoided. Heapsort demonstrated stable behavior but consistently slower execution compared to the other algorithms.

Performance on Reverse-Sorted Inputs:

For reverse-sorted data, the performance trends varied across input sizes.

- At n=5,000, Merge Sort (0.010413 s) outperformed both Heapsort (0.024312 s) and Quick Sort (0.018096 s).
- At n=20,000, Quick Sort (0.035183 s) was slightly faster than Merge Sort (0.038037 s), while Heapsort (0.066401 s) remained the slowest algorithm.

These results suggest that Merge Sort maintains stable performance regardless of input order. Quick Sort did not exhibit significant degradation, indicating that the chosen pivot strategy avoided worst-case behavior. In contrast, Heapsort demonstrated consistent but comparatively slower performance.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\19407\Desktop\PB\Algorithms and Data Structures\Assignment 4> python emperical_comparision.py
Input Size    Distribution     Algorithm        Execution Time(s)
-----------------------------------------------------------------
1000          random           Heapsort         0.004055
1000          random           MergeSort        0.002017
1000          random           Quicksort        0.001590
1000          reverse          Heapsort         0.003974
1000          reverse          MergeSort        0.002729
1000          reverse          Quicksort        0.003644
1000          sorted           Heapsort         0.002441
1000          sorted           MergeSort        0.001361
1000          sorted           Quicksort        0.001526
2000          random           Heapsort         0.006162
2000          random           MergeSort        0.005696
2000          random           Quicksort        0.003436
2000          reverse          Heapsort         0.004667
2000          reverse          MergeSort        0.004437
2000          reverse          Quicksort        0.003278
2000          sorted           Heapsort         0.005487
2000          sorted           MergeSort        0.002884
2000          sorted           Quicksort        0.003225
5000          random           Heapsort         0.017070
5000          random           MergeSort        0.012155
5000          random           Quicksort        0.011230
5000          reverse          Heapsort         0.016208
5000          reverse          MergeSort        0.009518
5000          reverse          Quicksort        0.012308
5000          sorted           Heapsort         0.021841
5000          sorted           MergeSort        0.008680
5000          sorted           Quicksort        0.008956
10000         random           Heapsort         0.031728
10000         random           MergeSort        0.026871
```

```
10000         random           MergeSort        0.026871
10000         random           Quicksort        0.018981
10000         reverse          Heapsort         0.031721
10000         reverse          MergeSort        0.018796
10000         reverse          Quicksort        0.020802
10000         sorted           Heapsort         0.036076
10000         sorted           MergeSort        0.017480
10000         sorted           Quicksort        0.018797
20000         random           Heapsort         0.071725
20000         random           MergeSort        0.072951
20000         random           Quicksort        0.043874
20000         reverse          Heapsort         0.071692
20000         reverse          MergeSort        0.036682
20000         reverse          Quicksort        0.047541
20000         sorted           Heapsort         0.075463
20000         sorted           MergeSort        0.038884
20000         sorted           Quicksort        0.041254
PS C:\Users\19407\Desktop\PB\Algorithms and Data Structures\Assignment 4>
```

3.3 Scaling Behavior:

As the input size increased from 1,000 to 20,000 elements:

- All three algorithms had a consistent growth with O(n log n).
- Runtime approximately of the algorithms was doubled or slightly more than doubled as n increased, aligning with theoretical expectations.
- Growth pattern of Heapsort was stable for all distributions.
- Merge Sort algorithm scaled reliably, showing consistent performance relative to the amount of data processed.
- Quick Sort scaled efficiently and achieved the fastest execution times.

The results indicate that none of the algorithms encountered worst-case scenarios, as evidenced by the absence of abrupt spikes in runtime.

3.4 Relation to Theoretical Analysis:

Heapsort

Heapsort guarantees a time complexity of O(n log n) in the best, average, and worst cases. The benchmark results support this theoretical guarantee, as its execution time remained consistent across sorted, reverse-sorted, and random input distributions. But Heapsort was slower than Quick Sort and Merge Sort because:

- More frequent comparisons and swaps
- Less cache-friendly memory access patterns

These practical factors explain why Heapsort, even though it has strong theoretical guarantees, was not the fastest algorithm in the experiments.

Merge Sort

Merge Sort also guarantees O(n log n) time complexity in all cases. The experimental results show that its performance remained stable across sorted, reverse-sorted, and random inputs. This consistent behavior is due to its divide-and-conquer approach, which does not depend on the initial order of the data. Although Merge Sort requires O(n) additional memory, this extra space did not significantly affect its runtime for the input sizes tested.

Quick Sort

Quicksort has an average time complexity of O(n log n), but in the worst case its complexity can increase to O(n^2). In the experimental evaluation, Quicksort showed the fastest performance on random inputs and did not display quadratic growth for sorted or reverse-sorted data. This indicates that the pivot selection method, such as using a randomized pivot, helped prevent highly uneven partitions.

In practice, Quicksort often performs very efficiently. Although it does not guarantee O(n log n) in every case without careful pivot selection, it typically benefits from smaller constant factors and more efficient memory access patterns. As a result, it frequently achieves better practical performance compared to other sorting algorithms.

Conclusion:

The empirical comparison demonstrates that:

- The performance for Heap Sort for all input types is O(n log n) but may be slower in practice because of higher constant factors.
- Merge Sort has a stable and consistent performance of O(n log n), irrespective of the input.
- In most scenarios, Quicksort remains the most effective choice, assuming the pivot selection strategy prevents the algorithm from reaching its worst-case complexity.

These results show that, although theoretical complexity gives a general guarantee of performance, the actual runtime in practice is affected by implementation choices, how memory is accessed, and the constant factors involved in the algorithm.

Priority Queue Implementation and Applications

Part A: Priority Queue Implementation

4.1 Priority Queue Implementation

A priority queue is a type of abstract data structure where every element has a corresponding priority. Elements with higher priority are handled before those with lower priority. In this study, a binary max-heap was employed to implement the priority queue, enabling efficient management of elements based on their priority levels.

Data Structure Selection

The priority queue is implemented using a binary heap structured as a Python list, employing an array-based representation. This approach was chosen for its simplicity and efficiency. In this design, parent-child relationships within the heap can be determined directly through index calculations, eliminating the need for explicit pointer references.

For an element stored at index i:

- Parent index → (i − 1) // 2
- Left child index → 2i + 1
- Right child index → 2i + 2

This structure supports efficient insertion and deletion operations while preserving the heap property.

Task Class Design

A Task class was developed to represent individual jobs managed by the priority queue–based scheduling system. The purpose of this class is to encapsulate all information relevant to scheduling while keeping the heap operations independent of task-specific details. By separating data representation from heap logic, the implementation becomes modular, easier to maintain, and extendable for more advanced scheduling policies.

A Task class was developed to represent individual tasks within the scheduling system. Each task contains the following attributes:

task_id: A unique identifier used to distinguish one task from another. This allows the scheduler to track execution order and verify correctness of the scheduling behavior.

priority: A numerical value indicating the importance or urgency of the task. The scheduler uses this value to determine execution order, where higher priority tasks are processed before lower priority tasks.

arrival_time (optional): Represents the time at which a task becomes available in the system. This attribute enables simulation of real-world scheduling environments and can be used to resolve ties between tasks with equal priority.

deadline (optional): Specifies the desired completion time of the task. Although not strictly required for priority-based scheduling, this field allows the implementation to be extended to support deadline-aware or real-time scheduling strategies.

The class also provides a readable representation of a task when printed, enabling clear observation of execution order during testing and simulation. Overall, the Task abstraction improves clarity, supports future enhancements, and ensures the scheduling system remains organized and scalable.

Heap Type Selection

A max-heap is selected for this implementation so that the task with the highest priority value is always stored at the root and extracted first. This matches a priority-based scheduling policy where larger priority values indicate more urgent tasks. As a result, the scheduler can always select the next task to execute in O(1) time (by reading the root), while insertion and removal remain O(log n).

4.2 Core Operations

insert(task)

The insert operation adds a new task to the heap. The task is initially placed at the end of the underlying list. A sift-up (heapify-up) procedure is then performed, during which the element is repeatedly compared with its parent and moved upward until the heap property is restored.

Time Complexity:

In the worst-case scenario, the inserted element may move from a leaf node to the root of the heap. Since the height of a binary heap is O(log n), the overall time complexity of the insert operation is: O(log n)

extract_max()

Since a max-heap is implemented, the extract_max operation removes and returns the task with the highest priority, which is located at the root of the heap. If a min-heap was used instead, the corresponding operation would be extract_min().

After the root element is removed, the last element in the heap is moved to the root position. A heapify-down (sift-down) procedure is then performed to restore the heap property.

Time Complexity:

The heapify-down process may traverse the height of the heap. Because the height of a binary heap is O(log n), the overall time complexity of the extract_max operation is: O(log n)

increase_key(task, new_priority)

This operation increases the priority of an existing task. After updating the priority value, the heap property may be violated with respect to the task's parent. To restore the heap property, the element is moved upward through a sift-up procedure.

Time Complexity:

In the worst case, the element may move up the height of the heap. Thus, the time complexity is: O(log n)

decrease_key(task, new_priority)

When the priority of a task is reduced, the heap property may be violated with respect to the task's children. A heapify-down procedure is applied to restore the heap structure.

Time Complexity:

In the worst case, the element may move down the height of the heap. Therefore, the time complexity is: O(log n)

is_empty()

The is_empty operation checks whether the heap contains any elements.

Time Complexity: O(1)

Code Implementation

The priority queue was implemented using a binary max-heap represented as a Python list. A Task class was designed to model individual tasks, including attributes such as task ID, priority, arrival time, and deadline. A PriorityQueueMaxHeap class was developed to implement the heap-based priority queue operations.

The heap property is preserved through two adjustment procedures: a sift-up operation performed after insertions and priority increases, and a heapify-down operation applied after extractions and priority decreases. To demonstrate priority-based scheduling, the simulation inserts multiple tasks into the queue and repeatedly extracts the task with the highest priority, thereby illustrating execution in descending order of priority.

The project structure is organized as follows:

- **priority_queue.py** contains:
    - The Task class
    - The PriorityQueueMaxHeap class, including the insert, extract_max, increase_key, decrease_key, and is_empty methods, as well as the internal heap maintenance logic

- **scheduler_simulation.py** contains:
    - A basic scheduling demonstration that outputs the execution order, with tasks processed from highest to lowest priority.

priority_queue.py

priority_queue.py > PriorityQueueMaxHeap > right

```python
class Task:
    def __init__(self, task_id, priority, arrival_time=0, deadline=None):
        self.task_id = task_id
        self.priority = priority
        self.arrival_time = arrival_time
        self.deadline = deadline

    def __repr__(self):
        return f"Task(ID={self.task_id}, Priority={self.priority})"


class PriorityQueueMaxHeap:
    def __init__(self):
        self.heap = []

    # ---------- Helper index methods ----------

    def parent(self, i):
        return (i - 1) // 2

    def left(self, i):
        return 2 * i + 1

    def right(self, i):
        return 2 * i + 2

    def is_empty(self):
        return len(self.heap) == 0

    # ---------- Heapify (Sift Down) ----------

    def heapify(self, i):
        largest = i
        left = self.left(i)
        right = self.right(i)

        if left < len(self.heap) and self.heap[left].priority > self.heap[largest].priority:
```

priority_queue.py

priority_queue.py > PriorityQueueMaxHeap > decrease_key

```
12  class PriorityQueueMaxHeap:
49      def insert(self, task):
50          """
51          Insert a new task into the heap.
52          Time Complexity: O(log n)
53          """
54          self.heap.append(task)
55          i = len(self.heap) - 1
56
57          # Sift Up
58          while i != 0 and self.heap[self.parent(i)].priority < self.heap[i].priority:
59              self.heap[i], self.heap[self.parent(i)] = self.heap[self.parent(i)], self.heap[i]
60              i = self.parent(i)
61
62      def extract_max(self):
63          """
64          Remove and return the task with the highest priority.
65          Time Complexity: O(log n)
66          """
67          if self.is_empty():
68              return None
69
70          root = self.heap[0]
71          last = self.heap.pop()
72
73          if not self.is_empty():
74              self.heap[0] = last
75              self.heapify(0)
76
77          return root
78
79      def increase_key(self, index, new_priority):
80          """
81          Increase priority of task at given index.
82          Time Complexity: O(log n)
83          """
84          if new_priority < self.heap[index].priority:
85              return
86
87          self.heap[index].priority = new_priority
88
89          while index != 0 and self.heap[self.parent(index)].priority < self.heap[index].priority:
90              self.heap[index], self.heap[self.parent(index)] = self.heap[self.parent(index)], self.heap[index]
91              index = self.parent(index)
92
93      def decrease_key(self, index, new_priority):
94          """
95          Decrease priority of task at given index.
96          Time Complexity: O(log n)
97          """
98          if new_priority > self.heap[index].priority:
99              return
100
101         self.heap[index].priority = new_priority
102         self.heapify(index)
```

The complete runnable source code is provided in the GitHub repository.

4.3 Scheduler Simulation

A basic priority-based scheduler was developed to demonstrate the practical application of the priority queue. Tasks with varying priority levels were inserted into the queue, and the scheduler repeatedly extracted the task with the highest priority. This process continued until the queue was empty, thereby simulating priority-driven task execution.

This demonstration confirms the following:

- Tasks are processed strictly according to their assigned priority values.
- The execution order is independent of the order in which tasks are inserted into the queue.
- Tasks with higher priority are executed before those with lower priority.

Source Code for Scheduler Simulation

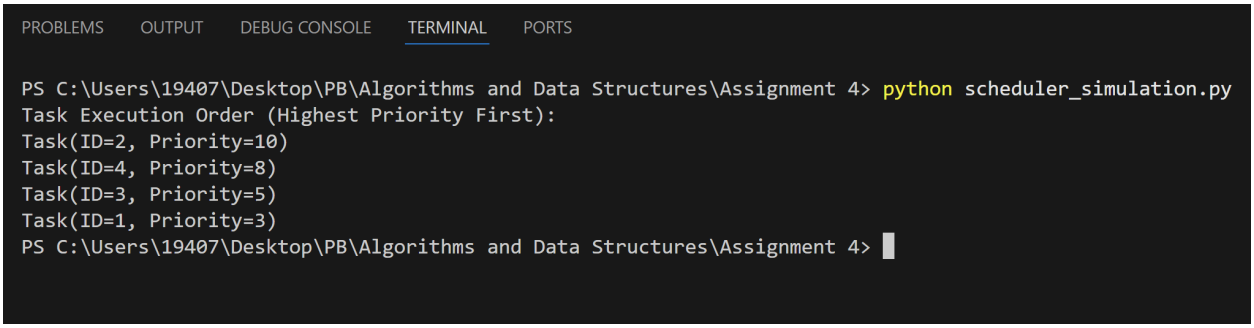
scheduler_simulation.py

```python
from priority_queue import Task, PriorityQueueMaxHeap

pq = PriorityQueueMaxHeap()

# Insert tasks
pq.insert(Task("1", 3))
pq.insert(Task("2", 10))
pq.insert(Task("3", 5))
pq.insert(Task("4", 8))

print("Task Execution Order (Highest Priority First):")

while not pq.is_empty():
    print(pq.extract_max())
```

The complete runnable source code is provided in the GitHub repository.

Output for Scheduler Simulation

The output demonstrates that tasks were executed in descending order of priority. The task with priority 10 was processed first, followed by tasks with progressively lower priority values. This result confirms the correctness and effectiveness of the heap-based scheduling mechanism.



GitHub repo – https://github.com/pranavibalakulla/MSCS532_Assignment4